The Automatic Parallelisation of Scientific Application codes using a Computer Aided Parallelisation Toolkit

C.Ierotheou, S.Johnson, P.Leggett, M.Cross and E.Evans
The Parallel Processing Research Group,University of Greenwich,London SE10 9LS,UK
H.Jin, M.Frumkin and J.Yan
NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, USA

Abstract

The shared-memory programming model is a very effective way to achieve parallelism on shared memory parallel computers. Historically, the lack of a programming standard for using directives and the rather limited performance due to scalability have affected the take-up of this programming model approach. Significant progress has been made in hardware and software technologies, as a result the performance of parallel programs with compiler directives has also made improvements. The introduction of an industrial standard for shared-memory programming with directives, OpenMP, has also addressed the issue of portability. In this study, we have extended the computer aided parallelisation toolkit (developed at the University of Greenwich), to automatically generate OpenMP-based parallel programs with nominal user assistance. We outline the way in which loop types are categorized and how efficient OpenMP directives can be defined and placed using the in-depth interprocedual analysis that is carried out by the toolkit. We also discuss the application of the toolkit on the NAS Parallel Benchmarks and a number of real-world application codes. This work not only demonstrates the great potential of using the toolkit to quickly parallelise serial programs but also the good performance achievable on up to 300 processors for hybrid message passing and directive-based parallelisations.

Introduction

The porting of applications to high performance parallel computers still remains a very expensive effort. The shared memory and distributed memory programming paradigms are two of the most popular models used to transform existing serial application codes to a parallel form. For a distributed memory parallelisation it is necessary to consider the whole program when using an SPMD paradigm. The whole parallelisation process can be very time consuming and error-prone. For example, data placement is an essential consideration to efficiently use the available distributed memory, while the placement of explicit communication calls requires a great deal of expertise. The parallelisation on a shared memory system is only relatively easier. The data placement appears to be less crucial than for a distributed memory parallelisation, but the parallelisation process is still error-prone, time-consuming and still requires a detailed level of expertise.

Despite the costly effort involved, the message passing-based parallelisation process for distributed memory architectures has tended to be favoured. This is largely due to the higher degree of scalability (often a characteristic of the architecture) and portability (provided by standardising the message passing library used e.g. MPI [1]). However, the porting of real application codes from machines that use a single serial processor to one with multiple processors is far from a trivial process irrespective of the paradigm or architecture being used. The relentless user desire for higher performance and scalability

together with the continuing evolution of parallel architectures has made the parallelisation and subsequent maintenance of a code a major programming effort [2, 3].

The re-emergence of the shared memory parallel machines typified by the cache-coherent Non-Uniform Memory Access architecture of the SGI Origin 2000 [4] has done much to promote the use of shared memory directives to describe parallelism in an application. In contrast to using message passing, the use of directives is relatively simple. For a single program multiple data (SPMD) parallelisation using message passing, consideration must be given to data placement (as the memory is physically distributed), masking of statements to ensure parallel execution and the introduction of communication calls to ensure comparable execution to the original serial code [5]. For a parallelisation based on loop distribution and using directives, consideration is only given to the loops and the visibility of variables. Another benefit to using directives is that they can easily be ignored since they are treated as comments if the compiler directive flag is not used. Therefore, the use of directives is generally less intrusive with fewer code modifications than that needed for a message passing-based parallelisation. Programming with directives is also relatively simple compared to writing message passing-based codes although it does not necessarily provide a performance benefit. In the worst case, the code will execute to give erroneous results if directives are incorrectly used and this can be tedious to debug, for example the errors may be symptomatic of run-time race conditions.

Ideally, one would like to be able to automatically insert directives (or message passing calls) into the original serial code with very little effort. In reality, this is not the case, the performance achievable for real-world industrial application codes using an automatic approach is largely dependent on the quality of the dependence analysis. Since many assumptions may be required due to the lack of knowledge (often available only from the user) this can significantly affect the quality of the generated code and hence the performance. Despite this limitation, many parallelising compilers have been developed over the years. Some of the more notable research and commercially available compilers have included Superb [6], Paraphrase [7], Polaris [8], Suif [9] and KAI's toolkit [10].

The focus of this paper is to look at the semi-automatic parallelisation of codes using an industry standard defining shared memory directives (OpenMP) as a means to describe the parallelism present in real-world scientific application codes.

OpenMP - an industry standard defining shared memory directives

The introduction of the shared memory directive standard, OpenMP [11], addresses the issue of portability across a range of platforms. The main aim of OpenMP is to achieve portability without significantly sacrificing the performance of the parallel execution. OpenMP includes a set of compiler directives and callable run-time library routines to support shared memory parallelism for the C, C++ and Fortran programming languages. To some extent, OpenMP will allow the programmer to incrementally develop a parallel implementation and this makes it more attractive as it is easier to program.

OpenMP follows the fork-and-join execution model so that each time a parallel region is defined the process is used. A brief description of the fork-and-join process is included here for completeness. At the start of the process a single "master" thread exists. The master thread executes sequentially until the first parallel construct (called `OMP PARALLEL`) is encountered. At this point the master thread creates a number of threads to assist the master thread in concurrently executing the statements in the parallel region. If a parallel loop is encountered (defined by `OMP DO`) then the iterations of the loop are distributed amongst all the threads. An implied synchronisation is performed at the end of the loop unless a `NOWAIT` directive option is specified. The `SHARED` and `PRIVATE` clauses at the start of the parallel or work-sharing constructs define if the data is visible globally or locally to a single thread. Reduction operations such as summations are handled in parallel by using the `REDUCTION` clause. At the end of the parallel region all the threads in the team synchronise and only the master threads continues with the program execution.

Optimisation of the directives and their placement is essential to generate parallel code that will execute efficiently. There is an overhead associated with every use of `OMP PARALLEL` so reducing the number of parallel regions (by fusing them together whenever legally possible) is a desirable optimisation. It is also the experience of the authors that the use of the `NOWAIT` clause (whenever this is legal) can significantly improve the parallel performance.

Semi-automatic parallelisation tools

The main goal for developing tools to assist in the parallelisation of serial application codes is so that as much of the tedious, manual and sometimes error-prone work is performed by the tools and in a small fraction of the time that would otherwise be needed for a manual parallelisation. With this in mind, the Computer Aided Parallelisation Toolkit has been developed over a number of years to enable the generation of generic, portable, parallel source code from the original serial code [12, 13, 14]. The toolkit generates SPMD based parallel code for distributed memory systems or loop distributed directive-based parallel code for shared memory systems. For distributed memory systems, the toolkit has been used to successfully parallelise a number of application codes [12, 15] based on the solution of a system of partial differential equations over a defined geometry using a mesh. The mesh over which these equations are solved is used as the basis for the partitioning of the data on to the distributed memory. The solution can be computed for a single block structured, unstructured or multi-zone structured meshes. The quality of the parallel source code generated benefits from many of the features provided by the toolkit. For example, the dependence analysis is fully interprocedural and value-based (i.e. detects the flow of data rather than just the memory location accesses)[16] and allows the user to assist with essential knowledge about program variables [17]. The placement and generation of communication calls also makes extensive use of the interprocedural capability of the toolkit as well as merger of similar communications [5]. Finally, the generation of readable parallel source code that can be maintained was seen as a major benefit. The use of the toolkit to generate parallel code for distributed memory systems will not be described in detail here since it has been documented elsewhere [5, 12, 16, 17].

The toolkit can also be used to generate parallel code with OpenMP directives from the original serial code. This approach also makes use of the very accurate interprocedural analysis and also benefits from a directive browser to allow the user to interrogate and refine the directives automatically placed within the code.

Automatic generation and placement of OpenMP directives in the serial code

The process the toolkit uses to automatically exploit loop level parallelism can be defined by three distinct stages (see [18] for more details of these stages and their implementation):

i. *Identification of parallel regions and parallel loops* – this includes a comprehensive breakdown of the different loop types (these are described in more detail below). Due to the current lack of support for nested parallel regions in OpenMP compilers, only the outermost parallel loops are considered for exploitation so long as they provide sufficient granularity. Since the dependence analysis is interprocedural, the parallel regions can be defined as high up in the call tree as possible, in doing so, provides a more efficient placement of the directives.

ii. *Optimisation of parallel regions and parallel loops* - the fork-and-join overhead (associated with starting a parallel region) and the cost of synchronising is greatly lowered by reducing the number of parallel regions required. This is achieved by merging together parallel regions where there is no violation of data usage. In addition, the synchronisation between successive parallel loops is possible if it can be proved that the loops can correctly execute asynchronously (using the `NOWAIT` clause).

iii. *Code transformation and insertion of OpenMP directives* – this includes the analysis for possible `THREADPRIVATE` common blocks due to the usage of the common block variables. There is also special treatment for private variables in non-threadprivate common blocks. If there is a usage conflict then a routine is copied and the common block variable is added to the argument list of the copied routine. Finally, the call graph is traversed to place OpenMP directives within the code, this includes the identification of `SHARED`, `PRIVATE` and `THREADPRIVATE` variable types.

An interactive browser to provide detailed information on loops

Although the dependence analysis carried out is very detailed, it can often contain dependencies that had to be assumed to exist. In these cases, user assistance can be used to improve the quality of the generated OpenMP code. This is done by classifying the different types of loops that generally exist in application codes and using a browser (Figure 1) to inspect and interrogate all the loops in turn. For example, the user can enforce the classification of a selected loop by re-defining the loop type. The user can also define the granularity threshold for a loop so that any loop below this level is not considered for parallelisation. In our study we have identified the following different types of loops:

i. *Totally serial loops* – These loops contain a loop-carried true data dependence that causes the serialisation of the loop i.e. data assigned in an iteration of the loop is used in a later iteration. (Other possible reasons for a loop to be defined as serial

include the presence of I/O or loop exiting statements within the loop body). The directive browser shows a list of the variables and a textual explanation of why the loop is serial. However, the data dependence may have been assumed to exist and the user may be able to supplement the dependence analyser with additional information to prove that the data dependence does not exist. Alternatively, the user may wish to enforce the removal of a serialising data dependence using the dependence browser (Figure 2) In addition, this loop type does not contain any nested parallel loops and also is not contained within a parallel loop.

ii. *Covered serial loops* – These are also serial loops containing a loop-carried true data dependence, so they can be treated in a similar way to totally serial loops. However, this type of serial loop is either nested within a parallel loop or contains parallel loops within it. In the latter case, if the serial loop can be made parallel (see *totally serial loops*) then the parallelism can be defined at a higher level and may therefore enhance the performance of the execution.

iii. *Falsely serial loops* – These loops are not serial due to a loop-carried true dependence. Instead, they will need to execute in serial due to the existence of pseudo dependencies that represent memory re-use as this needs to be considered when working within a globally addressable memory. The directive and dependence browsers can be used together with any additional information the user may wish to offer to re-examine if the variable(s) concerned can be privatised. In the process, dependencies into or out of the loop are examined to test if the variable could be made `PRIVATE`, or to re-examine if the loop carried pseudo dependencies are needed, in an attempt to allow the loop to execute in parallel.

iv. *Reduction loops* – The analysis is used to determine if the loop body computations represent a global reduction operation such as a `MAX` or summation. These loops provide a partial update of the results by each thread followed by a global update to give the final reduction value.

v. *Pipeline loops* – This is a special class of serial loops with loop-carried true dependencies. Directive-based software pipelines can be used to good effect in parallel. Figure 3 shows an example where OpenMP function calls are used to define the pipeline start-up before the `J`-loop and the pipeline shutdown after the loop. The example is taken from a version of the NAS APPLU benchmark. This is a similar strategy to that adopted for a software pipeline used in a distributed memory parallelisation with message passing. Figure 3 shows a software pipeline implementation using a high level message passing library called the Computer Aided Parallelisation Library (CAPLib) [19]. CAPLib is a thin layer that covers a choice of message passing libraries such as PVM, MPI, Cray Shmem etc.

vi. *Chosen parallel loops* – These are the parallel loops at which the `OMP DO` construct is defined. These loops may contain serial or parallel loops within their nesting but are not surrounded by other parallel loops.

vii. *Not chosen parallel loops* - Also parallel loops, but these have not been selected for application to the `OMP DO` directive. This is because these loops are surrounded by other parallel loops at a higher nesting level. In general, the OpenMP compiler suppliers do not currently support nested parallelism, therefore, even though parallelism exists at these lower levels, it is not currently exploited.

The accurate dependence analysis allows the algorithm to automatically generate efficient OpenMP code in many cases. In the experience of the authors, this typically leaves a small proportion of cases that require user interaction. For example, the use of workspace arrays is very common in application codes, but the value-based nature of the dependence analysis will often prove that no data is passed between iterations of a loop. The memory re-use (pseudo) dependencies must however be set. This correctly does not classify such loops as serial, however, the legal privatisation of these arrays to allow parallel execution requires that no data is passed into or out of these arrays from or to outside the loop. The value-based analysis, again greatly aids in proving that no such dependencies into or out of the loop exist.

Test cases

Parallelisation of the NAS Parallel Benchmark codes

The NAS Parallel Benchmarks were designed to compare the performance of parallel computers and have been widely used in this capacity. The details of the benchmarks and their message passing implementations can be found in [20] and [21], respectively. The dependence analysis was supplemented with very simple user information for some of the benchmark codes. More details on the parallelisation of these benchmarks using the toolkit can be found in [18] so only a brief report will be made here. Figure 4 shows the performance achieved for six of the NAS benchmark codes on an SGI Origin 2000 (R10000 CPU running @195MHz) for the class A size of problems. The comparisons show the performance for the hand tuned message passing (MPI-hand) and OpenMP (OMP-hand); the Computer Aided Parallelisation Toolkit using OpenMP (CAPO); and the SGI Power Fortran Analyser (SGI-PFA). The parallel code generated using the toolkit is not tuned for the Origin 2000 architecture, so that for example, there are no explicit 'optimisations' for cache usage/re-usage. A summary of the findings indicate that:

- It was possible to generate parallel code using the toolkit in a few minutes while the manually tuned parallelisations were created over a period of a few weeks.
- Code generated using the toolkit was within 5%-10% of hand tuned parallel performance
- Code generated by the SGI-PFA is not as efficient as that provided by the toolkit

Parallelisation of FDL3DI code (Air Force Research Laboratory)

The FDL3DI code was developed by M.Visbal at the Air Force Research Labs to study aeroelastic effects. The code solves the Navier-Stokes equations using a one-dimensional structural solver component. The parallelisation of this 10,000 line source code took approximately two hours (including user assistance) for the message passing-based parallelisation using a 2-dimensional decomposition and half an hour for the OpenMP-based parallelisation. The results shown in Figure 5 and Figure 6 are for a regular 100x100x100 node test case and indicate that very respectable performances were achieved with both message passing and directive based approaches. It is also important to recognise that the results are for the parallel code versions generated by the toolkit and that no manual optimisation has been performed. Table 1 shows a summary of the key communication requirements while Table 2 shows a summary of the key directives generated.

Parallelisation of the R-Jet code (Ohio Aerospace Institute)
The R-Jet code was developed by M.White and is a hybrid, high-order compact finite difference spectral method. It is used to simulate vortex dynamics and breakdown in turbulent jets. Although the code is explicit in time, the compact finite difference scheme requires the inversion of tri-diagonal matrix systems.
As part of the identification for directive placement, the algorithm automatically applied routine duplication to routines where there it was necessary to be able to fully exploit the parallelism present. The code fragment shown in Figure 7 shows a part of routine `rhs`, the two calls to `r2r` and a part of the routine `r2r`. The `J` loop in routine `rhs` and the `K` loop in `r2r` are both identified as being parallel and can therefore benefit from being encapsulated by the `OMP DO` construct. However, nested parallel regions are not currently fully supported by the vendors so one solution to exploiting the parallelism at both levels for different instances is shown in Figure 7. The complete list of routines duplicated can be seen in the call graph for the R-Jet code Figure 8.

Table 3contains a summary of the statistics for the OpenMP directives automatically generated by the toolkit. Figure 9 illustrates the execution performance of the automatically generated OpenMP directive-based parallel code for a 500x500 node test case. It demonstrates that a performance improvement of up to 32 processors of an SGI Origin 2000 was possible even for such a small test case.

Parallelisation of the INS3D code (NASA Ames)
There is a trend towards hybrid hardware systems that comprise clusters of nodes connected to each other through a communication interconnect. Within each node there is a number of processors and a common shared memory. One obvious scenario could be to exploit parallelism within a cluster using OpenMP directives while using message passing to communicate data between clusters. This multi-level exploitation of parallelism may have the potential to enable a more effective and scalable use of larger numbers of processors to solve a common problem. The Computer Aided Parallelisation Toolkit developed thus far has all the individual components to potentially exploit the hybrid systems. The strategy for combining these two approaches seems a natural extension. Indeed, a prototype has already been designed and implemented. However, care is needed to identify the applications where such a hybrid model can be used to good effect instead of using either pure message passing or pure OpenMP directives.

The parallelisation of the INS3D code using a mixed model of message passing and shared memory directives is shown as an example where such a model can be used effectively. A detailed account of this parallelisation was carried out by C.Kiris *et al* [22] but only a summary is included here. The INS3D code solves the 3D, incompressible Navier-Stokes equations and uses a structural, overset grid system. This is analogous to a multi-zone type application code. The manual MPI parallelisation was carried out by T.Faulkner and J.Mariani at NASA Ames and was used as the base code that was inputted to the toolkit. The toolkit was then able to complement the parallelism defined at the zone level by providing OpenMP directives for the parallelism defined within a zone (Table 4). The test case is the Space Shuttle Main Engine high pressure turbo-pump

impeller. The geometry was made up of 60 zones and 19.2 million grid points (the sizes of the zones ranged from 75,000 to over 1 million grid points). The results for the test case are shown in Figure 10 and demonstrate the impressive performance achievable for this hybrid parallelisation. The processors are arranged by MPI groups so that with 300 processors and 30 groups performing MPI/zone-level parallel execution, within each group there is a total of ten threads used to perform the OpenMP/intra-zone parallel execution.

Conclusions

The work presented here demonstrates a number of significant differences between the toolkit discussed here and other tools or compilers. It highlights the need for a very accurate dependence analysis including the detection of dependencies interprocedurally, and this is supplemented with the need for user interaction to aid in the parallelisation process. There is also a need to carefully insert directives in an efficient manner to exploit the systems as far as possible using generic techniques. Finally, this work has demonstrated the performance achievable when using the toolkit to parallelise real large scientific application codes.

Acknowledgements

The authors wish to acknowledge the assistance from C.Kiris (NASA Ames), P.Saddayapan (OSU) and R.Luczak (ASC) for their involvement in generating results for some of the test cases reported. The authors also wish to thank the many people at both Greenwich and NASA Ames who have helped in both the CAPTools and the CAPO developments. This work is supported by NASA Contract No. NAS2-14303 with MRJ Technology Solutions, No. NASA2-37056 with Computer Sciences Corporation.

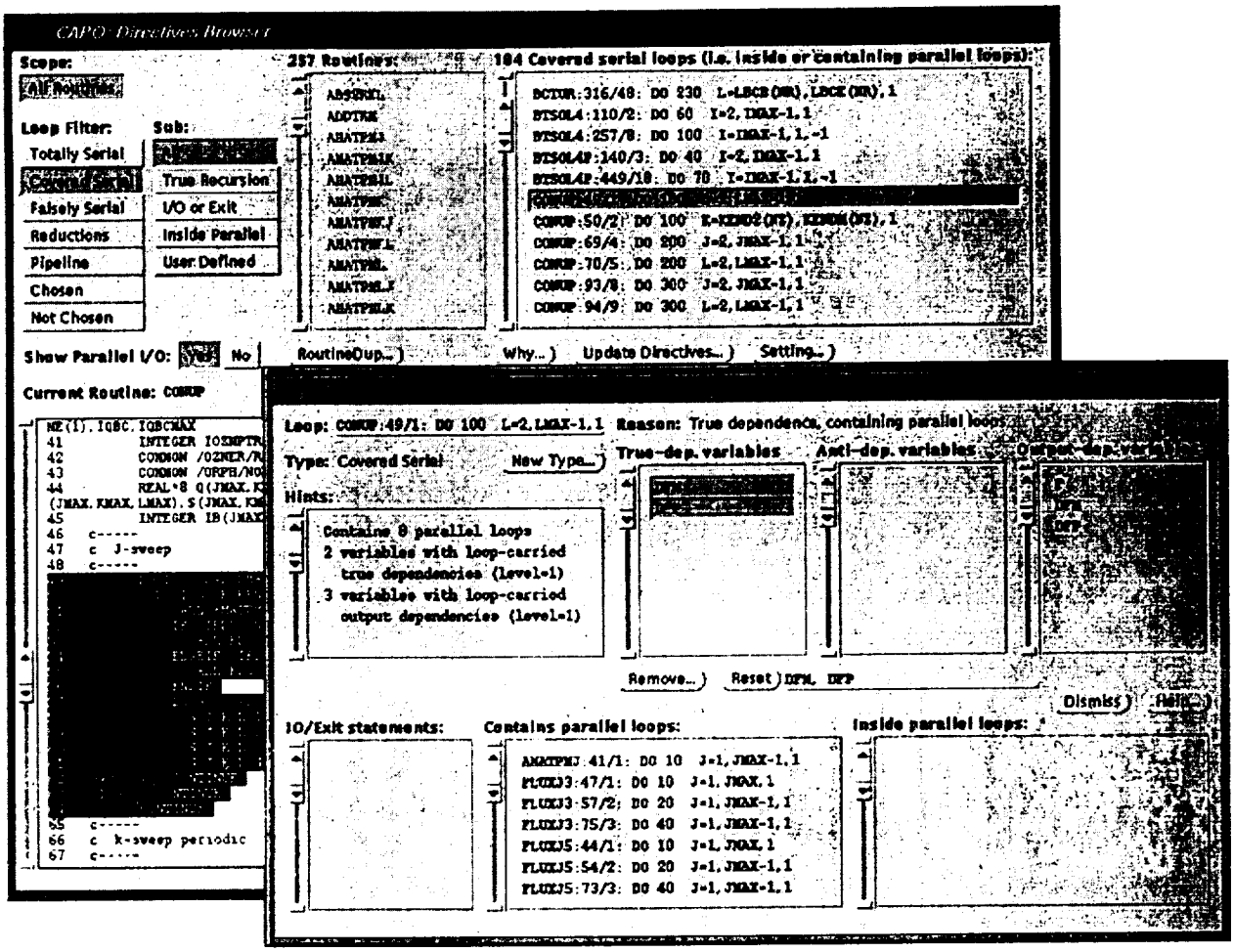


Figure 1 Browsers used to inspect all loop types in the application code and detailed information about the selected loop.

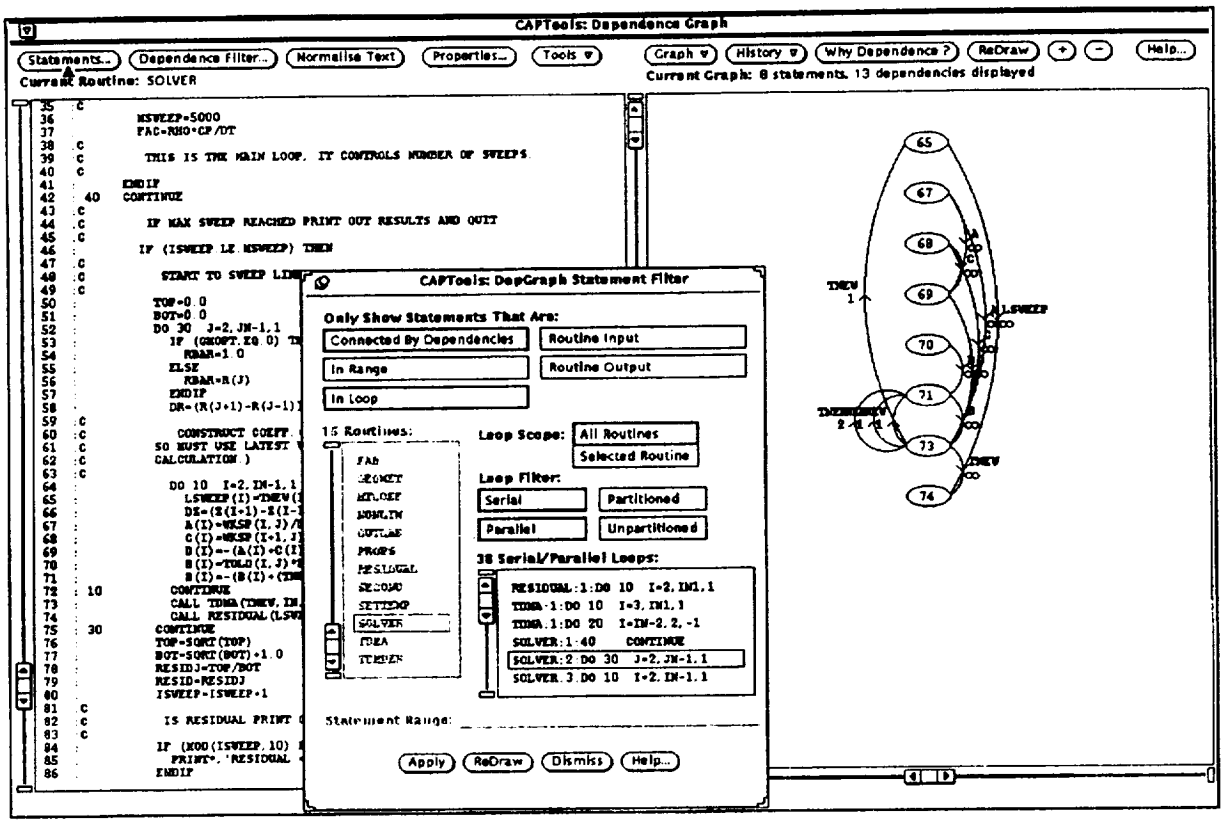


Figure 2 Dependence browser displaying the code and the equivalent dependence graph.

(a) Using OpenMP function calls to implement a software pipeline for routine BLTS

```
      lloop = jend-jst
      if (lloop .gt. mthnum) lloop = mthnum
      iam = omp_get_thread_num()
      if (iam .gt. 0 .and. iam .le. lloop) then
        neigh = iam - 1
        do while (isync(neigh) .eq. 0)
!$OMP FLUSH(isync)
        end do
        isync(neigh) = 0
!$OMP FLUSH(isync)
      endif
!$OMP DO SCHEDULE(STATIC)
        do j=jst,jend,1
          do i=ist,iend,1
c-------------------------------------------------------------------
c
c forward elimination and back substitution for diag. block inversion
c-------------------------------------------------------------------
          enddo
        enddo
!$OMP END DO nowait
      if (iam .lt. lloop) then
        do while (isync(iam) .eq. 1)
!$OMP FLUSH(isync)
        end do
        isync(iam) = 1
!$OMP FLUSH(isync)
      endif
```

(b) Using CAPLib message passing function calls to implement a software pipeline for routine BLTS

```
      CALL CAP_RECEIVE(v(1,2,LOW-1,k),nx0*5-10,3,CAP_LEFT)
        do j=MAX(jst,jst+LOW-2),MIN(jend,jst+HIGH-2),1
          do i=ist,iend,1
c-------------------------------------------------------------------
c
c forward elimination and back substitution for diag. block inversion
c-------------------------------------------------------------------
          enddo
        enddo
      CALL CAP_SEND(v(1,2,HIGH,k),nx0*5-10,3,CAP_RIGHT)
```

Figure 3 Implementation of a software pipeline using (a) OpenMP (b) message passing

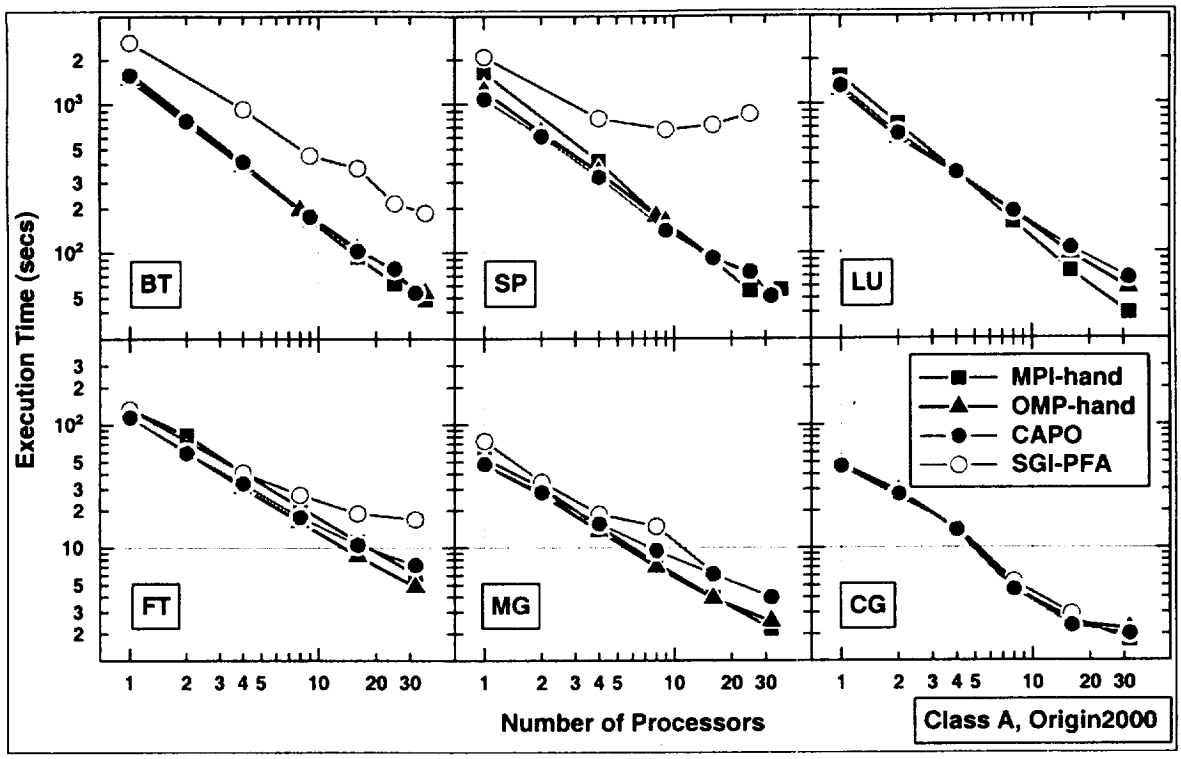
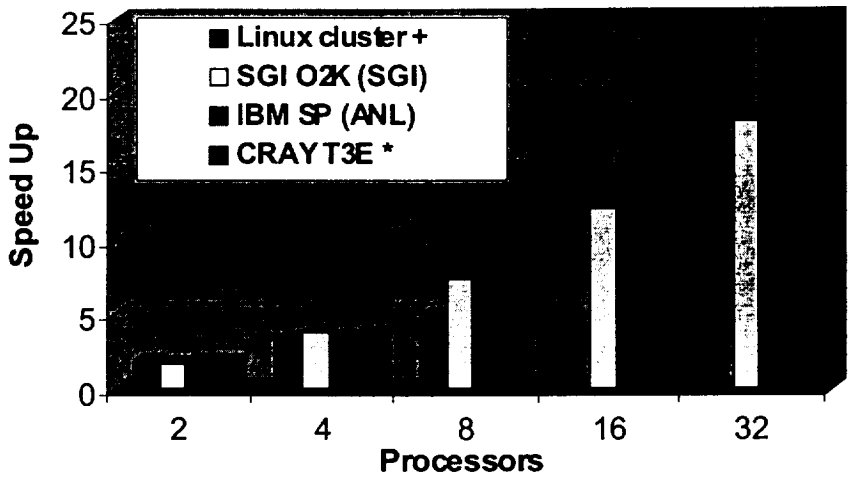


Figure 4 Various parallelisations of the NAS Parallel benchmark codes



* results shown for a small problem size + PC-based cluster using MYRINET

Figure 5 Performance of the message passing-based parallel FDL3DI code that was generated using the Computer Aided parallelisation toolkit.

Communication type	total
EXCHANGE	: 194
SEND/RECEIVE	: 72
BROADCAST	: 22
REDUCTION	: 20
PIPELINE	: 24

Table 1 Summary of communication types generated for the FDL3DI code as part of the message passing-based parallelisation using the Computer Aided parallelisation toolkit

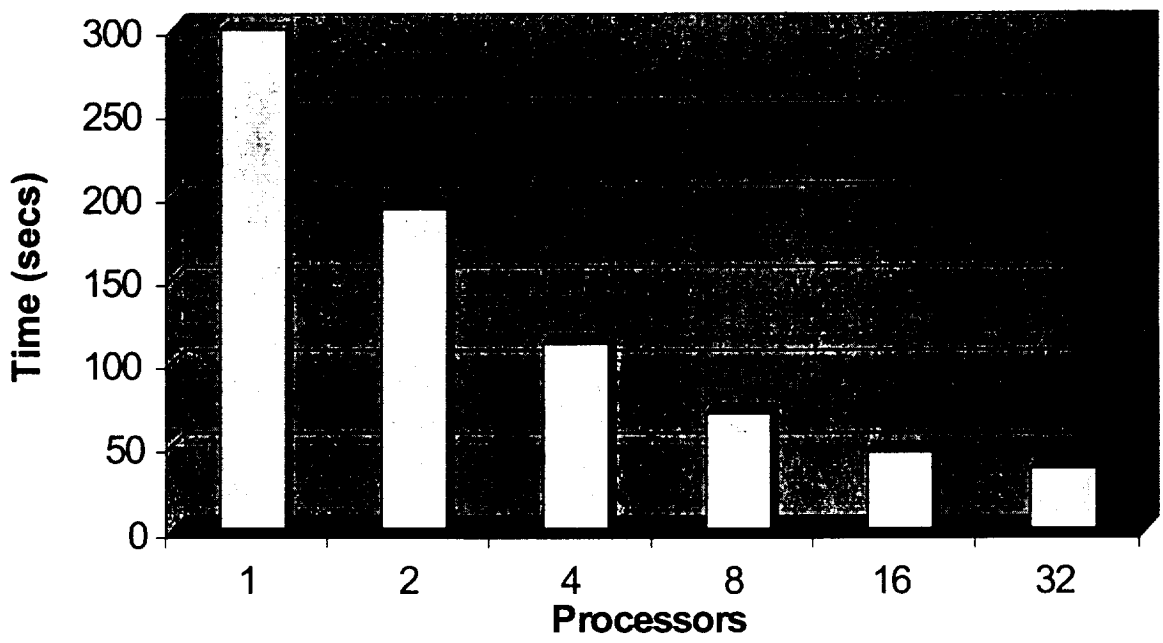


Figure 6 Performance on an SGI Origin 2000 of the OpenMP directive-based parallel FDL3DI code that was generated using the Computer Aided parallelisation toolkit.

Directive type	total
PARALLEL Regions	: 46
PARALLEL + DO Regions	: 43
Parallel DO Loops	: 194
ATOMIC/CRITICAL Sections	: 1
Regions with FIRSTPRIVATE	: 3
Regions with LASTPRIVATE	: 1

Table 2 Summary of directive types generated for the FDL3DI code as part of the OpenMP directive-based parallelisation using the Computer Aided parallelisation toolkit

Original serial code

```
call r2r(1)
do j=2,jmax
  call r2r(j)
enddo

subroutine r2r(j)
do k=1,kmax
   ...
enddo
```

Automatically generated parallel OpenMP code

```
      call r2r(1)
!$OMP PARALLEL DO DEFAULT(SHARED),PRIVATE(j)
      do j=2,jmax
        call cap_r2r(j)
      enddo
!$OMP END PARALLEL

      subroutine r2r(j)
!$OMP PARALLEL DO DEFAULT(SHARED),PRIVATE(k)
      do k=1,kmax
         ...
      enddo
!$OMP END PARALLEL

      subroutine cap_r2r(j)
      do k=1,kmax
         ...
      enddo
```

Figure 7 Automatic routine duplication to exploit parallelism at a number of levels

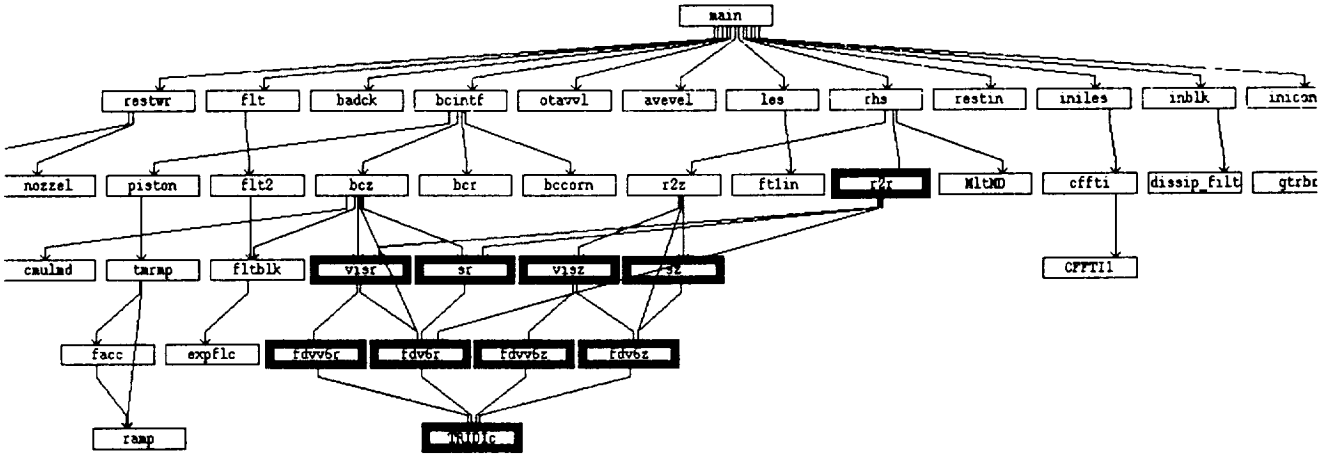


Figure 8 Call graph for the R-Jet code. Duplicated routines are shown highlighted

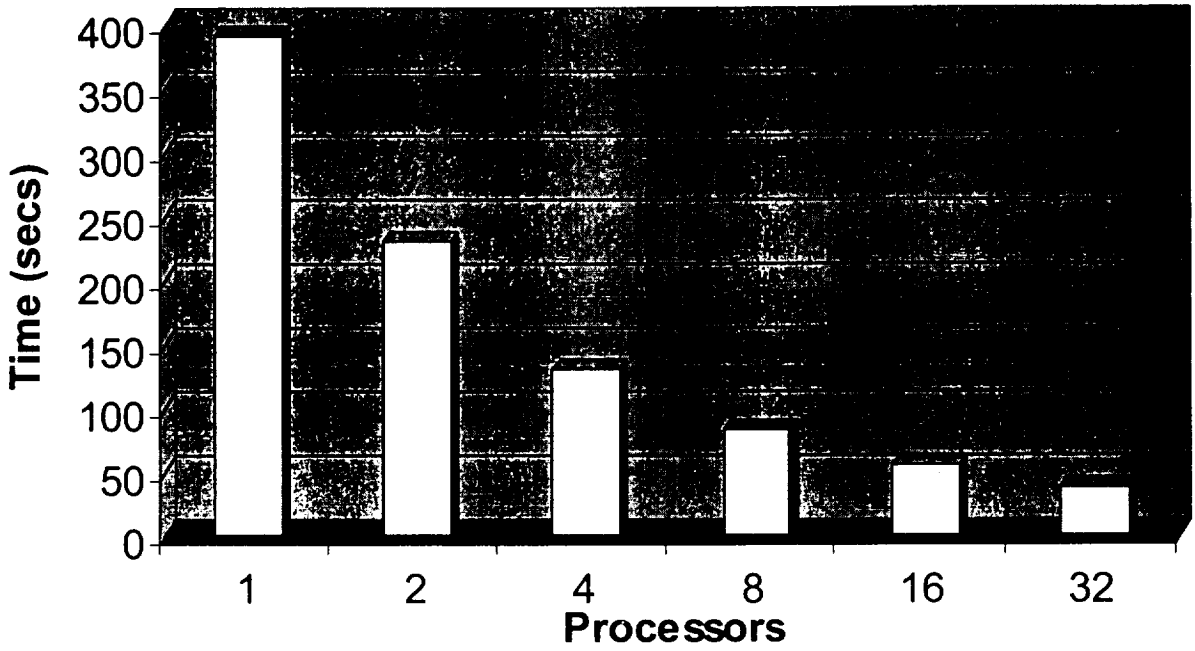


Figure 9 Performance on an SGI Origin 2000 of the OpenMP directive-based parallel R-Jet code that was generated using the Computer Aided parallelisation toolkit.

Directive type	total
PARALLEL Regions	: 9
PARALLEL + DO Regions	: 41
Parallel DO Loops	: 32
REDUCTION loops	: 4
Regions with FIRSTPRIVATE	: 1

Table 3 Summary of directive types generated for the R-Jet code as part of the OpenMP directive-based parallelisation using the Computer Aided parallelisation toolkit

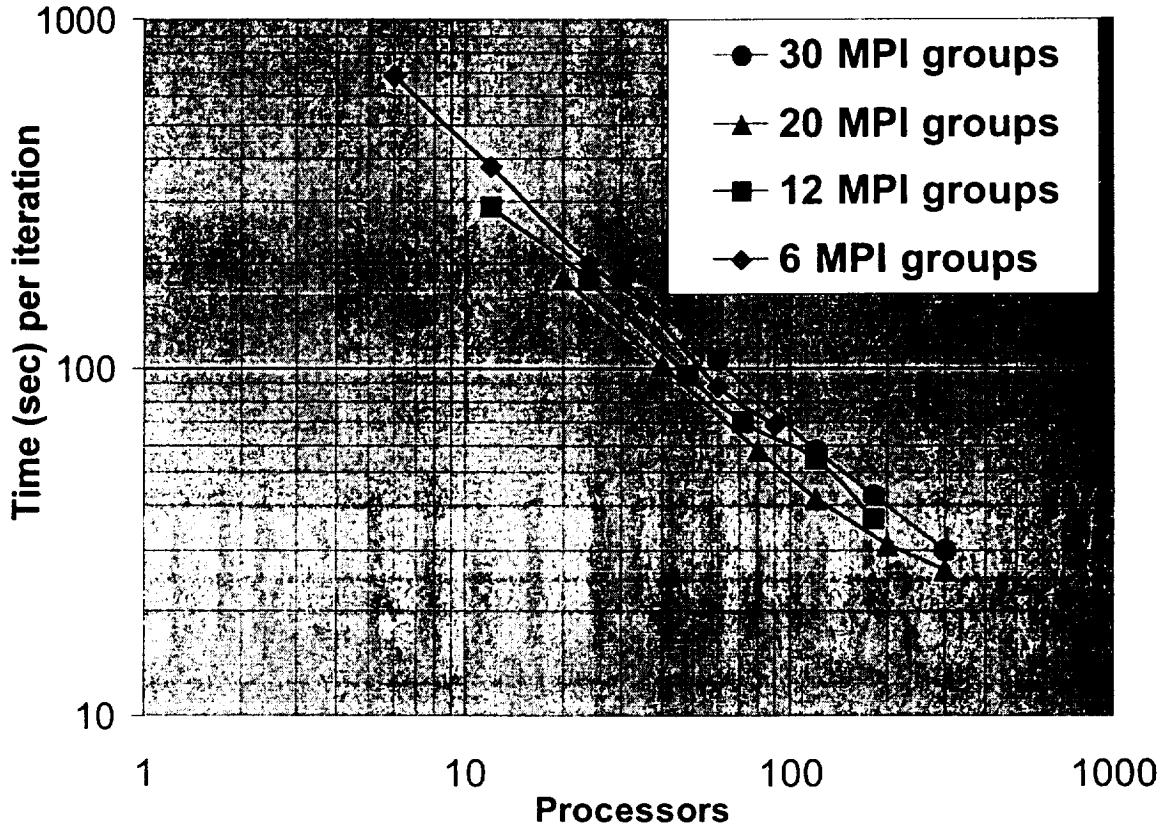


Figure 10 Performance of hybrid parallel code that includes MPI (performed manually at the zone level) and OpenMP (done using the toolkit and exploiting parallelism within a zone).

Directive type	total
PARALLEL Regions	: 95
PARALLEL + DO Regions	: 297
Parallel DO Loops	: 251
REDUCTION loops	: 79
ATOMIC/CRITICAL Sections	: 6
Regions with FIRSTPRIVATE	: 2

Table 4 Summary of directive types generated for the INS3D code as part of the OpenMP directive-based parallelisation using the Computer Aided parallelisation toolkit. (The code read into the toolkit was an MPI parallel version of the code)

References

1 William Gropp, Ewing Lusk, and Anthony Skjellum, Using MPI, 2nd Edition, MIT Press, 1992.

2 Johnson S P and Cross M, *"Mapping Structured Grid Three-Dimensional CFD Codes Onto Parallel Architectures"* Applied Mathematical Modelling, 15 1991.

3 Ierotheou C.S., Forsey C. and Block U. *"Parallelisation of novel 3D hybrid structured-unstructured grid CFD production code"* HPCN95, Springer-Verlag, 1995.

4 SGI Origin 2000 User guide, SGI, Mountain View, USA.

5 Johnson S.P., Ierotheou C.S. and Cross M. *"Automatic Parallel Code Generation For Message Passing on Distributed Memory Systems"* Parallel Computing, 22, 227-258, 1996.

6 Zima H P, Bast H -J, and Gerndt H M, *"SUPERB- A Tool for Semi-Automatic MIMD/SIMD Parallelisation"* Parallel Computing, 6, 1988.

7 Kuck *et al.*, *"The Structure of an Advanced Retargetable Vectorizer, Supercomputers: design and Applications Tutorial"* (Hwang K, ed) IEEE Society Press, Silver Spring MD, 1984.

8 Blume W., Eigenmann R., Faigin K., Grout J., Lee J., Lawrence T., Hoeflinger J., Padua D., Paek Y., Petersen P., Pottenger B., Rauchwerger L., Tu P., Weatherford S. *"Restructuring Programs for High-Speed Computers with Polaris, 1996 ICPP Workshop on Challenges for Parallel Processing"*, pages 149-162, August 1996.

9 Wilson R.P, French R.S,Wilson C.S, Amarasinghe S.P, Anderson J.M, Tjiang S.W.K, Liao S, Tseng C., Hall M.W, Lam M. and Hennessy J., *"SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers"* Computer Systems Laboratory, Stanford University, Stanford, CA

10 Kuck and Associates, Inc., *"Parallel Performance of Standard Codes on the Compaq Professional Workstation 8000: Experiences with Visual KAP and the KAP/Pro Toolset under Windows NT,"* Champaign, IL , Assure/Guide Reference Manual," 1997

11 OpenMP Fortran/C Application Program Interface, http://www.openmp.org/

12 Ierotheou C.S., Johnson S.P., Cross M. and Leggett P.F., *"Computer aided parallelisation tools (CAPTools) - conceptual overview and performance on the parallelisation of structured mesh codes"* Parallel Computing, 22, 197-226, 1996.

13 Evans E.W, Johnson S.P., Leggett P.F. and Cross M. *"The automatic code generation of asynchronous communications embedded within a parallelisation tool"* Parallel Computing, 23, 1493-1523, 1997.

14 Evans E.W., Johnson S.P., Leggett P.F., Cross M. *"Automatic and Effective Multi-Dimensional Parallelisation of Structured Mesh Based Codes"*. Parallel Computing, 26, 677-703, 2000.

15 Johnson S.P., Ierotheou C.S. and Cross M. *"Computer Aided Parallelisation Of Unstructured Mesh Codes"*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Editors H.R.Arabnia *et al*, publisher CSREA, vol. 1, 344-353, 1997.

16 Johnson S.P., Cross M. and Everett M. *"Exploitation of Symbolic Information In Interprocedural Dependence Analysis"* Parallel Computing, 22, 197-226, 1996.

17 Leggett P., Marsh A.T.J., Johnson S.P. and Cross M. *"User Interface Philosophy"* Parallel Computing, 22, 259-288, 1996.

18 Jin H., Frumkin M., Yan J. *"Automatic generation of OpenMP directives and its application to Computational Fluid Dynamics codes"*. Proceedings of International Symposium on High Performance Computing, Tokyo, p440, Japan, Oct. 16-19, 2000.

19 Leggett P.F., Johnson S.P. and Cross M. *"CAPLib – A 'Thin Layer' Message Passing Library to support computational mechanics codes on distributed memory parallel systems"*.

20 Bailey D., Barton J., Lasinski T., and Simon H. (Eds.), *"The NAS Parallel Benchmarks"*, NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, 1991.

21 Bailey D., Harris T., Saphir W., Van der Wijngaart R., Woo A., and Yarrow M., *"The NAS Parallel Benchmarks 2.0"*, RNR-95-020, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/

22 Kiris, C, Kwak D and Chan W., "Parallel Unsteady Turbopump Simulations For Liquid Rocket Engines". Proceedings of Supercomputing 2000, Dallas, Texas, 2000.